Agenda

- Asynchrony
- Callbacks
- Promises
- async/await
- Exercises
- Assignment?

Asynchronous JavaScript

"Keep every promise you make and only make promises you can keep."
 - Anthony Hitt

Asynchrony: Now & Later

synchronous vs *asynchronous*

In a *synchronous* programming model, things happen one at a time. When you call a function that performs a long-running action, it returns only when the action has finished and it can return the result. This stops your program for the time the action takes.

An *asynchronous* model allows multiple things to happen at the same time. When you start an action, your program continues to run. When the action finishes, the program is informed and gets access to the result (for example, the data read from disk).

Callbacks

What is a Callback?

- A callback function (say, Y) is a function that is passed to another function (say, X) as a parameter, and Y gets executed inside X.
- JS uses callback functions to handle asynchronous control flow

```
function onClick() {
   // Simulate a code delay
   setTimeout(() => {
       console.log(1);
   }, 500 );
}
const myInterval = setTimeout(function() {
   console.log(‘here’)
}, 1000);

document.addEventListener('click', onClick)

['a', 'b', 'c'].forEach(
      function(x) { console.log(x) }
)
```

Async callbacks

`addEventListener()`

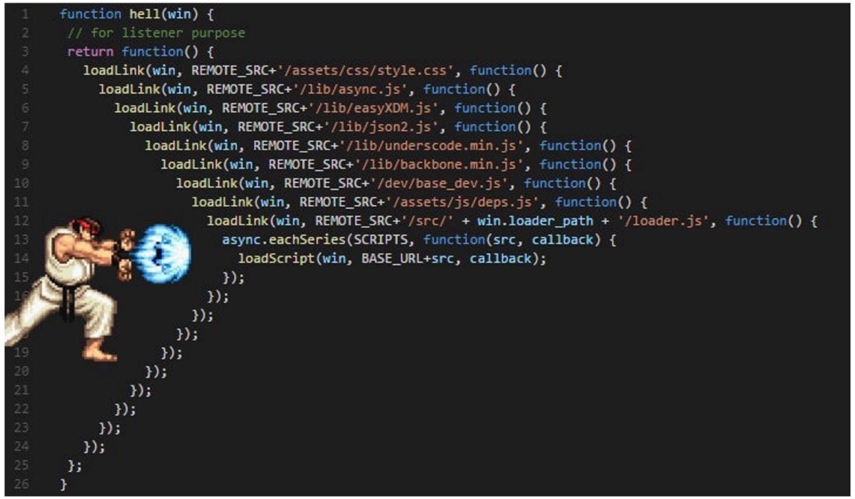
`XMLHttpRequest API/fetch`

`setTimeout()`

`setInterval()`

`requestAnimationFrame()`

Callback hell



```
function hell(win) {
 // for listener purpose
 return function() {
   loadLink(win, REMOTE_SRC+'/assets/css/style.css', function() {
     loadLink(win, REMOTE_SRC+'/lib/async.js', function() {
       loadLink(win, REMOTE_SRC+'/lib/easyXDM.js', function() {
         loadLink(win, REMOTE_SRC+'/lib/json2.js', function() {
           loadLink(win, REMOTE_SRC+'/lib/underscode.min.js', function() {
             loadLink(win, REMOTE_SRC+'/lib/backbone.min.js', function() {
               loadLink(win, REMOTE_SRC+'/dev/base_dev.js', function() {
                 loadLink(win, REMOTE_SRC+'/assets/js/deps.js', function() {
                   loadLink(win, REMOTE_SRC+'/src/' + win.loader_path + '/loader.js', function() {
                     async.eachSeries(SCRIPTS, function(src, callback) {
                       loadScript(win, BASE_URL+src, callback);
                     });
                   });
                 });
               });
             });
           });
         });
       });
     });
   });
 };
}
```

See the pyramid shape and all the }) at the end? Eek! This is affectionately known as **callback hell.**

Promises

The **frustration**

"JavaScript fatigue is what happens when people use tools they don't need to solve problems they don't have." –
Lucas F Costa

The **reality**

JavaScript is a ***synchronous*** programming language. But thanks to callback functions we can make it function like an ***asynchronous*** programming language.

What is a **promise**?

promise : noun : Assurance that one will do something or that a particular thing will happen.

So **what happens** when someone **makes a promise to you ?**

1. **A promise gives you an assurance that something will be done.**
Whether they (who made the promise) will do it themselves or they get it done by others is immaterial.

They give you an assurance, based on which you can plan something.

2. A promise can either be **kept** or **broken**.

3. When a **promise is <u>kept</u> you <u>expect</u>** something out of that promise.

You can make use of the output of a promise for your further actions or plans.

4. When a promise is **<u>broken</u>**, you want to know **<u>why</u>** (reason) the person who made the promise wasn’t able to keep up his side of the bargain.

Once you know the reason, and have a confirmation that the promise has been broken, you can plan what to do next.

5. When a **promise is made to us all we have is an assurance**.
We **can't** act on it immediately.

We can decide and formulate what needs to be done when the *promise is kept* (hence we have an expected outcome)
or *broken* (we know the reason and hence we can plan a contingency).

There are two parts to understanding promises.
Creating promises and *handling promises*.

Although most of our code will cater to handling promises created by other libraries, gaining a complete understanding is important.

Understanding the creation of promises will be increasingly important as you advance from the beginner stage.

What Is a Promise?

- The Promise object represents the eventual completion (or failure) of an asynchronous operation, and its resulting value.

```
new Promise(function(resolve, reject) {
   console.log('Initial');
   resolve();
})
.then(function() {
   throw new Error('Something failed');
   console.log('Do this');
})
.catch(function() {
   console.error('Do that');
})
```

Creating a promise

The constructor accepts a function called executor.
This executor function accepts two parameters: resolve and reject, which are in turn functions.

```
new Promise( /* executor */
function(resolve, reject) {
} );
```

Creating a promise

Promises are generally used for easier handling of asynchronous operations or blocking code, for example, file operations, API calls, DB calls, IO calls, etc.

These asynchronous operations initiate inside the executor function. If the asynchronous operations are successful the expected result is returned by calling the resolve function. Similarly, if there was some unexpected error the reasons are passed on by calling the reject function.

Creating a promise

Now we know how to create a promise. Let’s create a simple promise to help our understanding.

```
var keepsHisWord;
keepsHisWord = true;
promise1 = new Promise(function(resolve, reject) {
        if (keepsHisWord) {
                  resolve("The man likes to keep his word");
        } else {
                  reject("The man doesnt want to keep his word");
        }
});
console.log(promise1);
```

Creating a promise

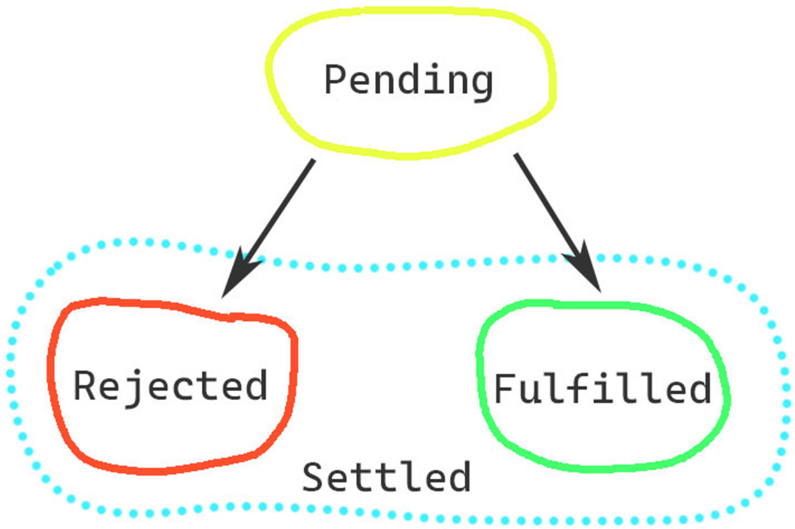
```
promise2 = new Promise(function(resolve, reject) {
setTimeout(function() {
resolve({
                message: "The man likes to keep his word",
                code: "aManKeepsHisWord"
                });
       }, 10 * 1000);
});
console.log(promise2);
```

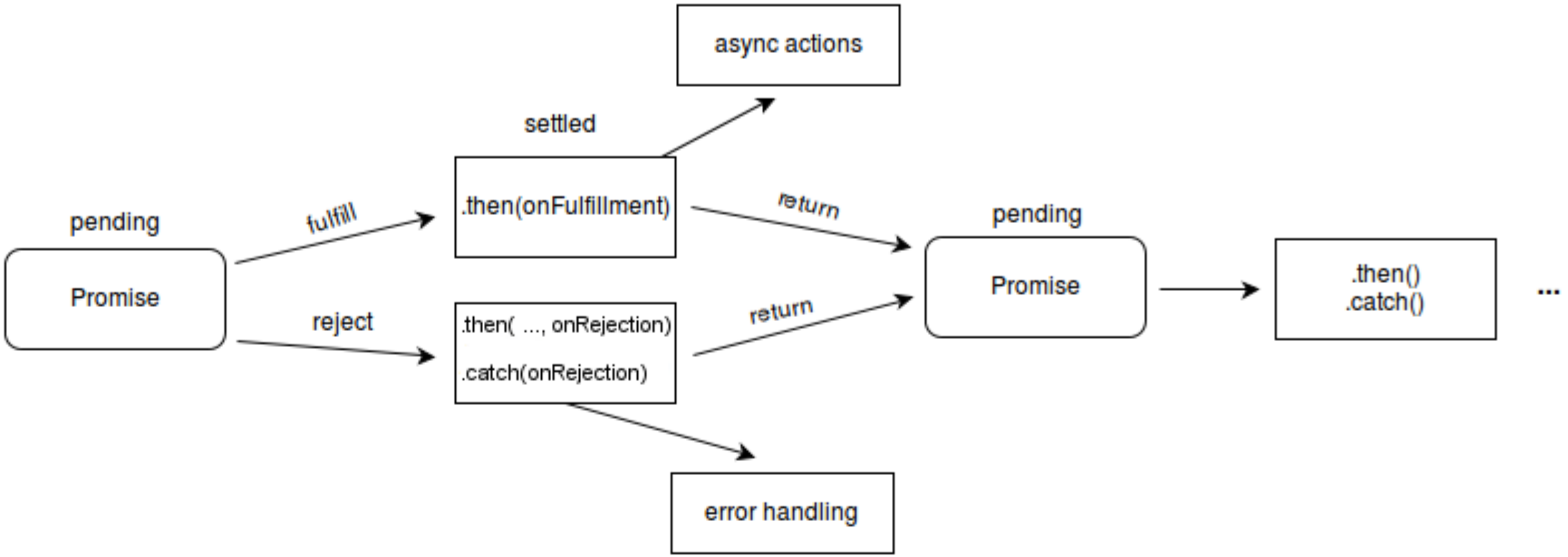
Creating a promise

```
keepsHisWord = false;
promise3 = new Promise(function(resolve, reject) {
if (keepsHisWord) {
resolve("The man likes to keep his word");
} else {
reject("The man doesn't want to keep his word");
}
});
console.log(promise3);
```

Promise states

- **pending**: initial state, neither fulfilled nor rejected.
- **fulfilled**: meaning that the operation completed successfully.
- **rejected**: meaning that the operation failed.




pending
Promise
fulfill
reject
settled
.then(onFulfillment)
.then(..., onRejection)
.catch(onRejection)
async actions
error handling
return
return
pending
Promise
.then()
.catch()
...

Guarantees

- Callbacks will never be called before the completion of the current run of the JavaScript event loop.
- Callbacks added with then(), as above, will be called even after the success or failure of the asynchronous operation.
- Multiple callbacks may be added by calling then() several times. Each callback is executed one after another, in the order in which they were inserted.

then Method

then sets **onFulfilled** and **onRejected** functions as handlers for our promise.

```
promise.then(
   onFulfilled?: Function,
   onRejected?: Function
) => Promise
```

Chaining

the **then()** function returns a **new promise**, different from the original:

```
doSomething()
.then(function(result) {
 return doSomethingElse(result);
})
.then(function(newResult) {
 return doThirdThing(newResult);
})
.then(function(finalResult) {
 console.log('Got the final result: ' +
finalResult);
})
```

Error propagation

If there's an exception, the browser will look down the chain for .catch() handlers or onRejected.

```
doSomething()
.then(
   result => doSomethingElse(result),
   error => console.log(error)
)
.then(
    newResult => doThirdThing(newResult)
)
.catch(failureCallback)
```

Creating a Promise around an old callback API

```
const wait = ms =>
   new Promise(
       resolve => setTimeout(resolve, ms)
   );

wait(10*1000)
   .then(
       () => saySomething("10 seconds")
   )
   .catch(failureCallback);
```

Composition

```
Promise.reject('error')
.then(value => console.log(value))
.catch( error => console.log(error))

Promise.all([ func1(), func2(), func3()])
.then(
   ([result1, result2, result3]) => {
       /* use result1, result2, result3 */
   }
);
```

Timing

To avoid surprises, functions passed to **then()** will never be called synchronously, even with an already-resolved promise:

```
Promise.resolve()
  .then(
    () => console.log(2)
  );
console.log(1);
```

async/await

What is *async/await?*

Async: special function that returns a promise

Await: pauses execution of an async function

```
async function main () {
   const result1 = await new Promise(
       resolve => setTimeout(() =>
resolve('async await'))
   )
   console.log(‘aaaa’)
   console.log(result1)
};

main()
```

Error handling

If a promise resolves normally, then await promise returns the result. But in the case of a rejection, it throws the error, just as if there were a throw statement at that line.

```
async function main() {
   try {
      let response = await fetch('http://...');
   }
   catch(err) {
      alert(err);
      // TypeError: failed to fetch
   }
}
main();
```

Excellent Bio

https://web.dev/promises/

https://blog.domenic.me/youre-missing-the-point-of-promises/

https://medium.com/dailyjs/javascript-async-await-zero-to-hero-plus-cheat-sheet-4b064401e29a